3 M3: Code lesen

3.1 Method Overloading

Mehrere Methoden können den gleichen Namen haben, wenn sie unterschiedliche Argumente haben. Dies wird ***Method Overloading*** genannt. Der Compiler erkennt anhand der **Signatur** (= Anzahl und Typen der Argumente), welche Methode ausgewählt werden muss. Dies müssen wir beim Code lesen manuell machen.

Aufgabe 1. Betrachte den folgenden Code:

```
1  public class Overloading {
2    public static int aufrunden(int n){
3      return n;
4    }
5
6    public static int aufrunden(double d){
7      if (d == (int) d){
8        return (int) d;
9      } else {
10       return (int) d + 1;
11     }
12   }
13
14   public static double aufrunden(double d, int digits){
15     d = d * Math.pow(10, digits);
16     d = aufrunden(d);
17     return d / Math.pow(10, digits);
18   }
19
20   public static void main(String[] args) {
21     System.out.println(aufrunden(3));
22     System.out.println(aufrunden(3.0));
23     System.out.println(aufrunden(3.001));
24     System.out.println(aufrunden(3.1111111, 2));
25   }
26 }
```

1. Programmfluss: Welche Zeilen werden in welcher Reihenfolge durchlaufen?
2. Stelle Call Stack und Stack Memory für den Programmablauf dar.
3. Was wird auf der Konsole ausgegeben?

Antwort:

1. 20-21 → (2-3) → 21-22 → (6-7-8) → 22-23 → (6-7-9-10) → 23-24 → (14-15-16 → (6-7-9-10) → 16-17) → 24-25
3. 3 3 4.0 3.12 (jede Zahl auf einer neuen Zeile)

3.2 Sichtbarkeit von Variablen

Variablen haben eine beschränkte Lebensdauer und sind nicht von überall her zugreifbar. Jede Variable hat einen **Sichtbarkeitsbereich *(scope)***. 'Sichtbar' heisst auch 'veränderbar' (ausser die Variable hat das Schlüsselwort `final`). Es gilt:

1. Variablen, die ausserhalb einer Methode definiert werden, sind aus jeder Methode heraus sichtbar. Wir nennen sie **global**.
2. Variablen, die innerhalb einer Methode definiert werden, sind auch nur innerhalb der Methode sichtbar. Wir nennen sie **lokal**. Dies gilt auch für die Argumente einer Methode.
3. Variablen, die innerhalb geschweifter Klammern (z.B. einer `if`- oder `while`-Anweisung) definiert werden, sind auch nur darin sichtbar.
4. Variablen, deren Sichtbarkeitsbereiche sich nicht überschneiden, dürfen den gleichen Namen haben.

Aufgabe 2.

```java
public class Visibility {
  static int s;

  static int y(int y) {
    int x = 1; /* 1 */
    if (y > 0) {
      int temp = x;
      x = y;
      y = temp;
    }
    s = 5;
    return x + y;
  }

  public static void main(String[] args) {
    int x = 2; /* 2 */
    int y = y(x); /* 3 */
    s = 4;
    y(y); /* 4 */
  }
}
```

1. Finde für jeden der vier Fälle eine Variable im obigen Code und nenne ihren Sichtbarkeitsbereich.
2. Stelle Call Stack und Stack Memory für den Programmablauf dar.
3. Fülle die Werte der Variablen zu verschiedenen Zeitpunkten in die Tabelle ein. Setze dabei `ND` für „nicht definiert“ und ∅ für „leer“.

	/* 1 */	/* 2 */	/* 3 */	/* 4 */
s				
x				
y				

Antwort auf Frage 1.
1. `s` (in der ganzen Klasse sichtbar)
2./4. `y` von Z. 4 und `x` von Z. 5 (beide sichtbar Z. 5-12)
bzw. `x` und `y` von Z. 16/17 (sichtbar von 16/17-19)
2. `temp` (nur sichtbar Z. 7-9)

3.3 Fehlersuche

Es gibt beim Programmieren drei Typen von Fehlern:

1. **Kompilierfehler *(compiling errors)***, die bereits der Compiler findet und einen Fehler ausgibt. Diese können sein:
   - Syntax-Fehler (Befehle falsch geschrieben, fehlende oder falsch gesetzte Klammern, Operatoren, Semikolons etc.)
   - Undefinierte oder mehrfach definierte Variablen.
   - Falsche Typen, z.B. wenn ein Befehl ein `int`-Argument erwartet, aber einen `double`-Wert übergeben bekommt.
2. **Laufzeitfehler (*run time errors*)**. Diese fallen erst beim Ausführen des kompilierten Programm auf und führen zu deren Abbruch mit Fehler. Beispiele:
   - Variablen, deren Inhalt zu unerlaubten Dingen führt wie `a / b` für `b = 0;` oder `Integer.parseInt(s)` für `s = "xyz";`.
   - Stack Overflow oder "Out of Memory"-Exception.
3. Probleme, die keine Fehlermeldung verursachen, aber trotzdem zu klar unerwünschtem Verhalten führen:
   - Integer overflow
   - U.U. Ein Programm, das nicht **terminiert** (sondern endlos läuft)

Aufgabe 3. Finde alle Kompilier-, Laufzeit- sowie weiteren Fehler im folgenden Programm (8 insgesamt):

```
1  public class Problems {
2
3    static void dont() {
4      panic();
5    }
6
7    static int panic() {
8      dont();
9    }
10
11   static void marvin(int n) {
12     System.out.println(n * n * n * n);
13   }
14
15   static void fortytwo(int six) {
16     seven = 6;
17     System.out.println(seven * six)
18   }
19
20   static void fordPrefect(int towel) {
21     System.out.println(4 / towel);
22   }
23
24   public static main(String[] args) {
25     dont();
26     marvin(100000000);
27     fortytwo(7.);
28     fordPrefect(0);
29   }
30 }
```

1. Kompilierfehler:
   Z. 7: Methode ist vom Typ `int`, gibt aber nichts zurück.
   Z. 16: Die Variable `seven` wurde nie definiert.
   Z. 17: Fehlendes `;`
   Z. 24: Die `main`-Methode muss einen Typ (nämlich `void`) haben
   Z. 27: Falscher Typ (`double` statt `int`) für das Argument `six`
2. Laufzeitfehler:
   Z. 21: Division durch Null nach Aufruf mit Argument 0 in Z. 28.
   Z. 25: Die Methoden `dont()` und `panic()` rufen sich endlos gegenseitig auf. Irgendwann ist der Call Stack voll, was einen Stack Overflow) ergibt.
3. Unerwünschtes Verhalten:
   Z. 12: Integer Overflow bei Aufruf mit Argument 100000000 in Z. 26.